# Implementing Wilson-Dirac Operator on the Cell Broadband Engine

Khaled Z. Ibrahim, François Bodin

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1880

# IMPLEMENTING WILSON-DIRAC OPERATOR ON THE CELL BROADBAND ENGINE

KHALED Z. IBRAHIM , F. BODIN

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

# Implementing Wilson-Dirac Operator on the Cell Broadband Engine

Khaled Z. Ibrahim , F. Bodin

Systèmes communicants
Projet CAPS

Publication interne n1880 — December 2007 — 23 pages

**Abstract:** Computing the actions of Wilson-Dirac operators consumes most of the CPU time for the grand challenge problem of simulating Lattice Quantum Chromodynamics (Lattice QCD). This routine exhibits many challenges to implementation on most computational environments because of the multiple patterns of accessing the same data that makes it difficult to align the data efficiently at compile time. Additionally, the low computation to memory access ratio makes this computation both memory bandwidth and memory latency bounded.
In this work, we present an implementation of this routine on Cell Broadband Engine. We propose runtime data fusion, an approach aiming at aligning data at runtime, for data that cannot be aligned optimally at compile time, to improve SIMDized execution.
We also show DMA optimization technique that reduces the impact of BW limits on performance. Our implementation for this routine achieves 31.2 GFlops for single precision computations and 8.75 GFlops for double precision computations.

**Key-words:** IBM Cell BE, Vectorization, SIMD, Lattice QCD, Parallel Algorithms, Wilson-Dirac

*(Résumé : tsvp)*

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

# Mise en œuvre de l'opérateur de Wilson-Dirac sur l'architecture Cell

**Résumé :** La mise en œuvre de l'opérateur de Wilson-Dirac est l'un des calculs les plus coûteux de QCD sur réseau. Ce calcul pose de nombreux challenges de mise en œuvre liés à l'accès aux données. En effet les multiples patrons d'accès rendent difficile l'optimisation de l'alignement des données pour des accès mémoire efficaces. Par ailleurs, le ratio calcul / accès mémoire engendre une saturation de la bande passante mémoire qui limite l'exploitation des unités de calculs.
Ce rapport présente une mise en œuvre sur l'architecture Cell de l'opérateur de Wilson-Dirac. Des techniques de fusion de données à l'exécution sont proposées pour résoudre le problème des contraintes d'alignement de données et l'utilisation des opérateurs SIMD des unités de calcul du Cell. De plus, une technique permettant d'optimiser les transferts DMA est aussi décrite. Notre implémentation atteint 31.2 Gflops en calcul simple précision, 8.75 Gflops en calcul double précision.

**Mots clés :** IBM Cell, Vectorisation, SIMD, QCD sur réseau, Algorithme parallèle, Wilson-Dirac

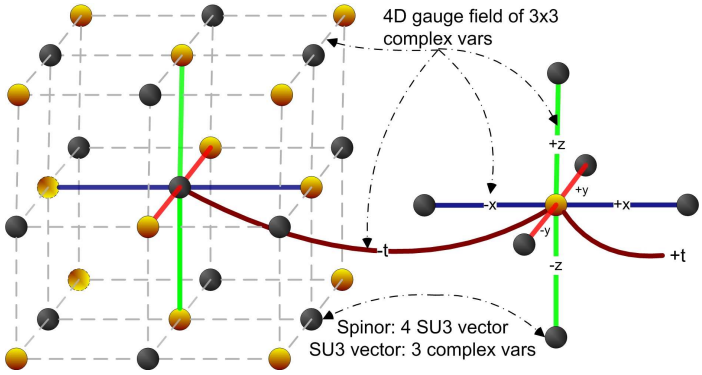


Figure 1: 4-dimensional space time lattice QCD.

# 1 Introduction

Efficient implementation for computing the action of Wilson-Dirac operators is of critical importance for the simulation of lattice Quantum Chromodynamics (Lattice QCD). Simulating Lattice QCD aims at understanding the strong interactions that binds quarks and gluons together to form hadrons. In lattice QCD, a four-dimensional space-time continuum is simulated, where quantum fields (quarks) are symbolized at the lattice sites and quantum fields (gluons) are symbolized at the links between these sites. Lattice spacing should be small to obtain reliable results which requires enormous amount of computations. Figure 1 shows the discretization of the four-dimensional space-time space of the lattice QCD.

The use of accelerator for scientific computing has always been experimented by many researchers. Among recently attractive technologies are graphic processing units (GPU) and Cell Broadband Engine. The Lattice QCD community, as well as other high performance computing communities, started exploring the possibility of using these accelerators to build cost effective supercomputers to simulate these problems.

Using GPU, for instance, has been investigated [1, 2], especially with the advent of general purpose programming environment such as Cuda [3] for graphic cards. The main challenge in these environments is the over-protection that most manufacturers adopt to hide their proprietary internal hardware design.

The use of Cell broadband engine is also under consideration of many Lattice QCD groups. An analytical model to predict the performance limits of simulating lattice QCD is developed [4]. Some simplified computation was also ported to the cell [5]. These studies affirmed the fact that the computation of lattice QCD is bandwidth limited (or memory bound) and tried to predict the performance of a real implementation.

In this study, we introduce an implementation of the main kernel routine for simulating Lattice QCD. In this implementation, we tried to provide answers to two main questions;

the first is how to SIMDize the computation in an efficient way; the second question is how to distribute the lattice data and how to handle memory efficiently.

For efficient SIMDization, we introduce the notion of runtime data fusion to align data at runtime that cannot be aligned optimally at compile time. Furthermore, while allocating lattice data on the main memory, we introduce analysis for data on the frames level to create optimized DMA requests that removes redundancy of data transfers as well as improves contiguity of memory accesses.

The rest of this report is organized as follows: Section 2 introduces the Cell broadband architecture and the software development environment. Section 3 introduces the Wilson-Dirac computation kernel. The SIMDization problem is tackled in Section 4. Section 5 details the proposed memory layout and the analysis leading to optimizing the memory transfers. We comment on the utilization of the Cell BE on Section 6. Section 7 concludes this report.

# 2 Cell Broadband Engine and Software Development Environment

In this study, we target developing efficient implementation of the main kernel routine of simulating Lattice QCD on Cell Broadband Engine (BE). We used IBM Cell BE SDK 3.0 [6]. We explored our implementation on current Cell BE as well as the future generation Cell with enhanced double precision (EDP)[1].

We used the simulator provided by the SDK to analyze the performance of our implementation and we verified the performance, except for Cell EDP, on a IBM BladeCenter® QS20 system with dual-Cell BE processors (running at 3.2 GHz).

Figure 2 outlines the basic component of the Cell BE processor. The Cell BE chip is composed of multiple heterogeneous cores; a PowerPC compatible master processor (with dual SMT) (PPE) and eight synergistic processing elements (SPE).

The execution unit on the PPE can handle control flow intensive codes while the execution unit on the SPE is optimized to handle SIMD computations. Each SPE has a 128 register file, each 16-bytes wide. The SPE has a small (256 KB) special memory called *local store* that execution unit can access with a pipelined latency of one cycle.

The main data is usually stored in the external memory and data are transferred back and forth with memory through special DMA APIs. Each SPE has two pipelines, one is specialized mainly on doing integer and floating point operations (even pipeline) and the other is specialized mainly in doing shuffling, branching and load/store operations (odd pipeline).

[1] For Cell EDP, the performance numbers are just estimates based on information collected from the simulator.

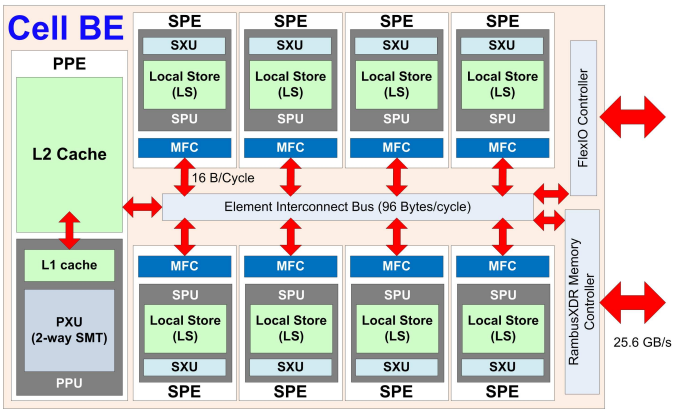


Figure 2: Cell Broadband Engine

# 3 Wilson-Dirac Operator

In this study, we ported the computation of the actions Wilson-Dirac on spinor field based on the code of the ETMC collaboration, see for instance [7, 8].

Computing the actions of Wilson-Dirac operator is the most time consuming operation in simulating lattice QCD. Equation 1 details the computation of the actions of Wilson-Dirac operator. This computation involves a sum over quark field ($\psi_i$) multiplied by a gluon gauge link ($U_{i,\mu}$) through the spin projector ($I \pm \gamma_\mu$).

$$\chi_i = \sum_{\mu=\{x,y,z,t\}} \kappa_\mu \left\{ U_{i,\mu} \left(I - \gamma_\mu\right) \psi_{i+\hat{\mu}} + U^{\dagger}_{i-\hat{\mu},\mu} \left(I + \gamma_\mu\right) \psi_{i-\hat{\mu}} \right\} \tag{1}$$

Where $\gamma_x = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$, $\gamma_y = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$, $\gamma_z = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$,

$\gamma_t = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$, $I$ is a unity matrix, and $\kappa_\mu$ represents the hopping term.

The representation of each gauge field is a special unitary SU(3) matrix ($3 \times 3$ complex variables). The spinors are represented by four SU(3) vectors composed of three complex variables. The routine implementing this computation is called Hopping_Matrix. Parallelization of the routine involves dividing the lattice into two dependent subfields odd and even, as shown in Figure 1. Each spinor of the odd subfield is surrounded by spinors of the even subfield and vice versa. The computation sweeps on spinors from one subfield making the other subfield temporarily constant, thus breaking data dependency.

Even though Equation 1 shows regular computation across all sites of the lattice, the computation usually faces the challenge of the low ratio of floating point operations to memory references. This makes this computation memory bandwidth and latency bounded.

In Section 4, we discuss the problem of efficiently SIMDizing this code, while the data alignment and communication is investigated in Section 5.

# 4 SIMDizing Wilson-Dirac Computations on Cell Broadband Engine

The main problem that prevents SIMDizing this code efficiently is the different patterns of accessing the same data, due to the spin projector in Equation 1, that make no single representation optimal at runtime. Each gauge field SU(3) matrix is accessed twice (positive and negative directions). The computation may involve the original matrix or the conjugate transpose of the matrix. The matrix is usually stored with only one representation. The problem is exacerbated for spinors because each spinor is accessed in eight different contexts depending on the space direction. Each access involves different spinor vectors and operations alternating between vector addition, subtraction, conjugate addition, and conjugate subtraction.

Aligning data such that all these operations are performed optimally at the same time is not possible. Different earlier approaches for SIMDizing this code align the data on one layout and then use shuffle operations to change alignment of the data at runtime to perform the needed computations.

Another problem is that data are represented by $3 \times 3$ complex matrix and 4 $3 \times 1$ complex vectors, which do not map perfectly to power of 2 data alignment. For the cell processor data should be aligned in the 16 Bytes boundary to be efficiently accessed. DMA is also better aligned in 128 bytes boundary.

In this work, we define "Runtime Data Fusion" as a solution for the above problems of data alignment. We show that the performance can be greatly improved using this technique.

Efficient SIMDization of the code requires alignment of the data in a way that reduces the dependency between instructions, reduces the number of shuffle instructions, and allows efficient instructions like multiply-add to be executed.

## 4.1 Runtime data fusion

Two conventional representation of complex structure are commonly used; the first combines the real and the imaginary parts into one structure; the second separates the real and the imaginary part into two separate arrays. Figure 3 shows these data layout for storing a $3 \times 1$ complex vector and $3 \times 3$ matrix for double precision aligned into 16 bytes word.

In Table 1, we list the number of instructions that are needed to do a matrix-vector multiplication. We consider the average for doing vector-matrix multiply and vector- transposed conjugate matrix multiply. For the first representation, the real and the imaginary

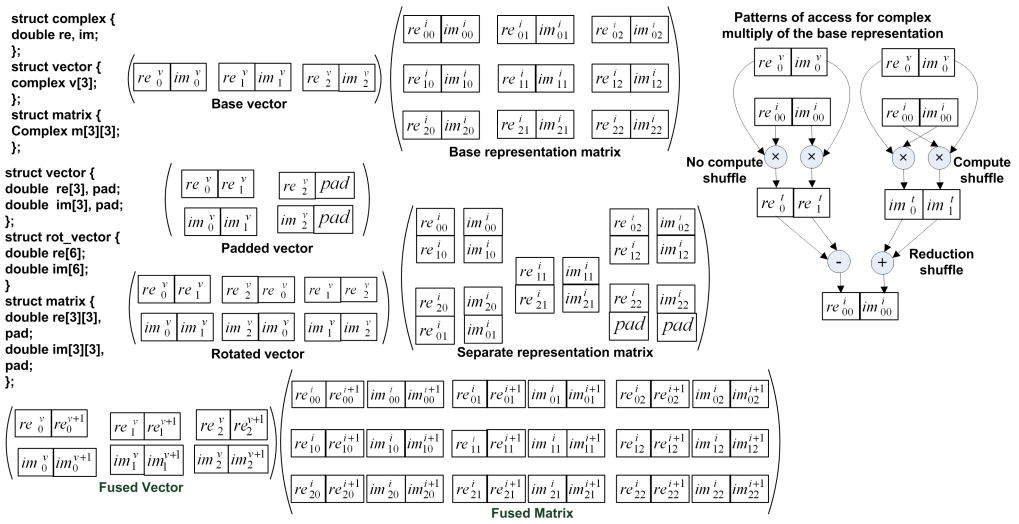


Figure 3: Multiple representations of structures based on complex variables aligned on 16 bytes word. The top shows merged representation, the middle shows separate representation, and the bottom shows the fused representation.

part faces different treatment, thus requiring shuffles. The second representation separates the real part and imaginary parts into two separate arrays, which involves additional shuffles for transposing the matrix.

The computational requirements based on these alignments favor separating the arrays for the real part and the imaginary part. The number of floating points is reduced because of the possibility to use multiply-add and multiply subtract instructions. These instructions cannot be used with the first layout because the real and imaginary parts share the same 16 bytes word.

Even with the larger number of instructions, the first layout is favored by the perfect alignment within the boundary of the word. The second layout requires either to use padding (25% of the total space for spinors and 10% of the gauge field link) or the data will not be perfectly aligned. Increasing the size of the data because of alignment can severely reduce the performance of this application because it is bandwidth limited as will be detailed in Section 5. Aligning data not to 16 bytes boundary will severely penalize loading and storing data and will nullify the computation benefit achieved by the reduced instructions. We adopted the first alignment of complex data as base for comparison, especially that the original implementation (optimized for SIMDization on Intel SSE2) adopted it throughout the whole code for simulating Lattice QCD.

Considering the fused version in Figure 3, the number of floating variables can be reduced significantly and no shuffling is needed except at the fusion/disjoin stages. The fusion introduced Figure 3 shows the two-way matrix fusion for complex variables of double precision.

| | Merged | Seperate | Fused |
|---|---|---|---|
| add, sub | 18 | 12 | 6 |
| madd, msub | 0 | 10 | 9 |
| mul | 22 | 10 | 9 |
| compute/reduce shuffles | 31 | 14 | 0 |
| transpose/conjugate shuffles | 3 | 5 | 0 |
| fuse shuffles | 0 | 0 | 6 |

Table 1: Instruction decomposition for vector-matrix multiply on SPE. The count represent the average for vector-by-matrix and vector-by-transposed matrix.

The fused matrix removes the need for shuffling in case of conjugate access to the array elements because complex and real variables are aligned in separate 16 Bytes boundary. Transposing or conjugating the matrix does not involve additional shuffles.

This fused alignment is unfortunately not possible at compile time, especially for spinors because it requires having a unique order of accessing spinor at compile time. In other word, given a spinor $i$ we need to determine a unique spinor that will always precede it in computation and a unique spinor that will always follow it. The surrounding spinors in every access context can be different and it will be a very large space overhead to keep multiple coherent copies.

Because of the performance associated with data fusion and difficulty of doing it statically, we consider the following proposal for "runtime data fusion"

1. Data are fused at runtime for a number of structures dependent on the number of elements per memory word, which usually involve some startup shuffle operations.

2. Optimized kernel of computation is written assuming fused data. Fused data are kept alive in registers as long as they are needed.

3. The final results for the optimized kernel (output spinors) are then disjoined back before storing them to the memory.

The steps involved in code transformations to support runtime data fusion are shown in Figure 4. Our technique involves fusing unrolled code to align data that can not be aligned statically due to the multiple access patterns encountered at runtime. The fusion process involves grouping data that will be accessed with the same pattern of access on SPU word size (*i.e.*, aligning them in 16 bytes boundary). For single precision computation, 4-bytes floating points, spinors are computed in a group of 4 output spinors. Consequently, the input gauge fields and the input spinors are combined into groups of four (4-way fusion). For double precision, optimal alignment requires fusing the computation for two output spinors (2-way fusion).

Runtime fusion adopts the same data structure used for the base. No alignment problem is encountered. The fused version only exists during computation, living in registers.

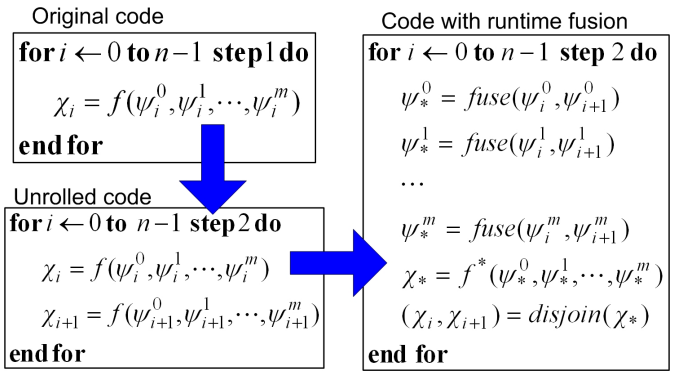


Figure 4: Code transformations for "runtime data fusion" computation.

The main feature of the Cell SPE that allows this technique is the large register file. Merging data structure in the beginning of computation incurs minimal overhead if all fused data are kept in registers as long as they are needed for computation. Runtime fusion can prove difficult for other processors with SIMD instruction set, that have small register count, for instance Intel SSE. We need to keep not only input fused data but also intermediate results alive in registers, especially that data are not accessed frequently. During the computation of a group of two spinors in double precision, almost 6 KB of memory are accessed while the register file can hold only 2 KBytes. Knowing that some registers are needed to hold shuffle patterns, intermediate results, and other bookkeeping operations, careful register lifeness analysis of registers is needed to minimize the possibility of spilling the register file to the memory. We did this analysis on a basic block size ranging between 2-2.4 Kilo instructions. In our implementation, we managed to use almost 110 of the SPE 128-bit registers without the need to spilling fused data or intermediate results to the memory.

Figure 5 shows the dynamic instruction decomposition for four implementations of the base complex alignment and the fused version. We have two computation flows in terms of the shuffle needed, one for the double precision and the 2-way single precision, and the other kernel is for the 2-way double precision and 4-way single precision. Perfect fusion kernel, 4-way single and 2-way double, provides better chance of reducing shuffles of data. The shuffling is reduced to less than 37% of the original shuffle operations count for single precision when we use 4-way fusion and less than 22% for double precision. Going for no fusion for single precision, not presented, will incur additional overheads during transposing matrixes and would provide a much worse performance.

Figure 5 shows also reductions in floating point operations, in addition to the reduction in shuffle instructions, because runtime fusion of data allows using multiply-add or multiply-subtract more frequently as clarified earlier in Table 1 .

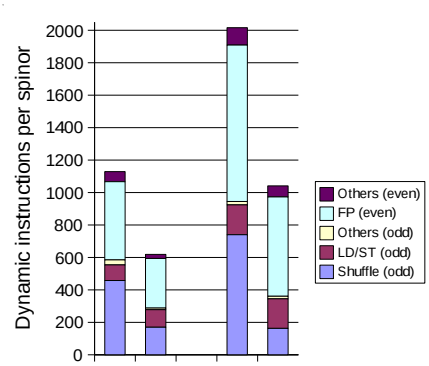


Figure 5: Decomposition of dynamic instructions per spinor computation. Two fusion level are shown for both single precision and double precision computations.

The impact on average execution cycles per spinor computation is detailed in Table 2. For these performance cycles, we aligned all data on the local store of Cell SPE. The execution cycles were reduced significantly for the versions with fusion compared with the versions with less or no fusion. For single precision the reduction is 35% for 4-way fusion compared with 2-way fusion. For double precision, the reduction is 40% on current generation Cell BE and is predicted to be 53% for future generation Cell EDP.

In Table 2, we define memory instructions efficiency as the percentage of the minimum load/store instructions, needed by the computation, to the actual load/store instructions executed during computation. As shown in the table, memory instructions efficiency is very high (ranging from 97.3% to 99%) in our implementation. Careful register lifeness analysis is needed to achieve this but is also facilitated by the large register file available on the Cell BE that makes it possible to hold intermediate computation of the spinors data structure after fusion until the end of the computation. Only single precision with 4-way fusion has additional memory operations associated with the fuse/disjoin phase, reducing the memory instructions efficiency to 83%. Still the overall performance with 4-way fusion is much better that 2-way fusion.

Table 2 also shows the performance in GFlops for these implementations. Apparently, if the data are requested from the memory system outside the Cell BE, then the memory subsystem will not be able to afford these bandwidths[2]. Only double precision on current generation Cell BE requires moderate bandwidth because the execution of double precision computation is severely penalized during the issue stage of instruction execution. This table

[2]Padding is added for single precision gauge field matrix to make the number of element even (less than 4% aditional overhead).

| | cycles per spinor | GFlops | Bytes /FP | GB/s | Cycles/ Frame | LD/ST Effi- ciency |
|---|---|---|---|---|---|---|
| 2-way single on Cell | 794 | 51.84 | 0.935 | 48.73 | 50816 | 98.28% |
| 4-way single on Cell | 517 | 79.62 | 0.935 | 74.85 | 33088 | 82.7% |
| double Cell | 7741 | 5.32 | 1.79 | 9.52 | 247712 | 97.79% |
| 2-way double Cell | 4681 | 8.79 | 1.79 | 15.74 | 149792 | 99% |
| double Cell EDP | 1757 | 23.43 | 1.79 | 41.94 | 56224 | 97.79% |
| 2-way double Cell EDP | 824 | 49.96 | 1.79 | 89.42 | 26368 | 97.28% |

Table 2: Execution cycles for single and double precision computation.

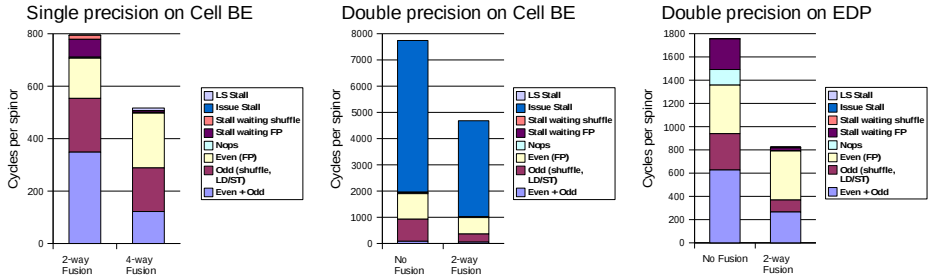


Figure 6: Kernel routine execution time breakdown for one spinor based on single precision and double precision computations on Cell BE and the future Cell EDP architectures.

shows that performance, in general, will be bounded by bandwidth; which justifies our choice of avoiding implementations based on data alignment that requires large paddings.

Efficient handling of memory decides the limits of the achievable performance on Cell architectures for Lattice QCD. This topic will be explored in details in the next section.

Figure 6 shows the decomposition of the execution cycles at runtime. For double precision on Cell BE the performance is dominated by the issue stall. The performance improvement for 2-way fusion is attributed to reduction in the number of issued FP (due to using multiply-add and multiply-subtract as one instruction instead of two separate instructions). For single precision on Cell BE and double precision on future generation Cell EDP, the shuffling is reduced (fewer cycles in the critical path of the execution time is needed by the odd pipeline and fewer stalls on shuffles). Additionally some nops are removed of the critical path of execution on future generation Cell EDP with double precision computation.

The unoverlapped odd pipeline cycles (implying stall of the even pipeline, responsible for FP operations) is reduced by 20%, 63%, and 66% for single precision on Cell BE, double precision on Cell BE, and double precision Cell EDP, respectively.

# 5 Lattice QCD Memory Alignment

Traditionally, the computation of Wilson-Dirac routine is parallelized by aligning part of the lattice near a computing element and the results of computation from this computing element are communicated with other computing elements. Fixing lattice per local store has the advantage of reducing the DMA requests with the memory. Applying the same model on the Cell processor is challenged by the following:

- The local store associated with SPE is very small. The number of output spinors that can be computed in this memory is no more than 64 double precision spinors or 128 single precision spinor. This leads to trivially small sublattice size.
- The computation-to-communication of small lattice is severely small. The results from on each local store will need to be communicated not only to the other SPE sharing the chip but also to the SPE for the other Cell chips. The need for simulating lattice of millions of spinors leads to the need for thousands of Cell processors making the slow communication a dominating factor for performance.
- The synchronization between the SPEs will be very frequent. In Table 2, we show the number of cycles needed to finish one frame of computation of 64 single precision spinors or one frame of 32 double precision spinors. Computing a frame requires cycles in the order of tens of thousands. These cycles leaves trivially small amount of inter-communication time and cannot scale well on a parallel machine. In our experiments, we noticed that the variation in execution time due to the wait for DMA is very large which makes any frequent synchronization on this architecture ineffective. Synchronization causes contention on resources because all SPEs will be either acquiring memory, doing computation or waiting at synchronization point.

The conventional approach to program Cell BE is to store the data on the memory system, then to bring frames of data for processing. Each SPE takes responsibility of doing the computation for part of the dataset. To compute one spinor, the data communicated with the external memory is 1504 bytes (assuming 16 bytes alignment) for single precision computations and 2880 bytes for double precision. This leads to Bytes/FP equals to 0.935 and 1.79 for single precision and double precision, respectively. Considering solely the bandwidth restriction of the cell memory system at 25.6 GB/s, the upper limit on performance will be 27.4 GFlops for single precision computation and 14.3 GFlops for double precision.

Putting data in the main memory solves the limited storage size of the SPE, but direct application of this approach will assign SPE computing threads the job of requesting data to operate upon. The data need to be aligned in contiguous memory regions to facilitate fetching.

As explained earlier, each spinor appears in the computation of eight other spinors. The context of access (relative access with other spinors) is not the same for each of these spinor access scenarios. Because of coherence we cannot have multiple copies easily for spinors; instead we need to compute indices of spinor location before fetching them. An implementation of this approach on Cell will face the following obstacles:

- It is extremely inefficient to do address calculation in the SPE because of the need for control flow and integer operations for many spinor sites. These computations cannot be easily SIMDized.
- It is not possible to compute indices of spinors and store them on the local store because of the limited size of the local store. Alternatively, storing them on the main memory will add the additional step of fetching indices from the main memory in the critical path of execution, in addition to stressing the system scarce bandwidth.
- Computing individual spinor location then requesting it will be associated with DMA fragmentation which can severely impact the performance and reduces the effective bandwidth observed during execution.

Because we know that storing spinors in the main memory is the solution suitable traditionally for Cell BE and since we know that there is redundancy on accessing the data (especially for spinors), we analyzed the data available within frames targeting the following objectives:

- Reducing the stress on the scarce bandwidth.
- Improving contiguity of DMA requests (having fewer DMAs).

The next section shows that frame analysis can lead to achieve these goals at least in part.

## 5.1 Contiguity analysis of the data space

The data accessed during spinors computation mostly belongs to the gauge field and spinors. Accessing each of these data structures has different attributes, as follows:

- For gauge field
  1. Each gauge link is surrounded by two spinors.
  2. The gauge field is not updated during the computation of the Wilson-Dirac operator. Consequently, each gauge link can have multiple copies in separate locations because no coherence is needed between these copies.
  3. The gauge field can be reordered arbitrary to improve the performance. Because two spinors are surrounding each link then preserving contiguity with respect to the surrounding spinors will require at most two copies for each gauge link.
  4. The spinors surrounding a gauge field always belong to two separate spinor subfields (one in the odd field and the other on the even field). One of these spinors is updated while the other is considered constant (belonging to the constant subfield). During a sweep (even-to-odd or odd-to-even) each gauge field link appears only once.

Based on the above, the best way to access the gauge field is to replicate the gauge field based on the spatial locality for each sweep of computation. This alignment can be done at compile time. Two gauge field organizations can be created, one is optimized for contiguity during the odd-even sweep of computation and a redundant copy is optimized for the even-odd sweep. In one sweep, no redundancy of data exists and data are organized in way to guarantee contiguity of access.

- For spinor field

  1. Each spinor is surrounded by eight gauge links. Each spinor is accessed in eight different contexts.
  2. Half the spinors are updated in each sweep of computation.

Multiple copies of the same spinor will require coherent update for all copies which would add load to the potentially overloaded memory system.

For spinors, we considered analyzing the frames of computation for contiguity and redundancy. Figure 7 shows part of the indices accessed in one frame of spinors. Computing an output spinor requires accessing one row of the input spinors. Looking at the indices of each row (needed to compute one spinor) we find no spatial locality. Looking across neighboring spinors, we observe some contiguity for the spinors streamed from the same directions of the space, for instance the columns for +t, -t, +x, and -x.

We can issue DMA requests from the memory based on their space alignment to improve contiguity of access from the external memory. Knowing that the local store is not arranged like cache, the non-contiguous access during execution of spinors on the local store is fortunately not penalized.

Contiguity of access favors issuing DMAs based on column major ordering from the main memory while accessing spinors in a row-major fashion from the SPE local store.

The second observation is that some of the spinors within the frame are repeated. To save bandwidth, we can bring the non-redundant part of the frame and then use the very fast local store to local store transfers. This can reduce the stress on the memory system and reduce the effect of bandwidth on performance. We similarly search for redundant data that can be moved between consecutive frames. Effectively, two copy lists are created; the first is used for copying redundant spinors within the same frame; and the second is used to copy spinors between contiguous frames.

Because the values of spinors changes during computation, while their locations remain fixed, the analysis to create optimized DMA lists and to create copy list is needed only once at the start of the program execution. This job is done more efficiently on PPE because it is control flow intensive. The PPE can also use the addresses of the spinors within the SPE local store so that absolute indexing is given back to the SPE to improve SPE runtime performance.

The SPE receives the addresses where its optimized DMAs are located in the start of creating its thread. In our implementation, we allocate a buffer for information about DMAs for 64 frames in the local store (occupying about 32 KB). Each frame represents the data

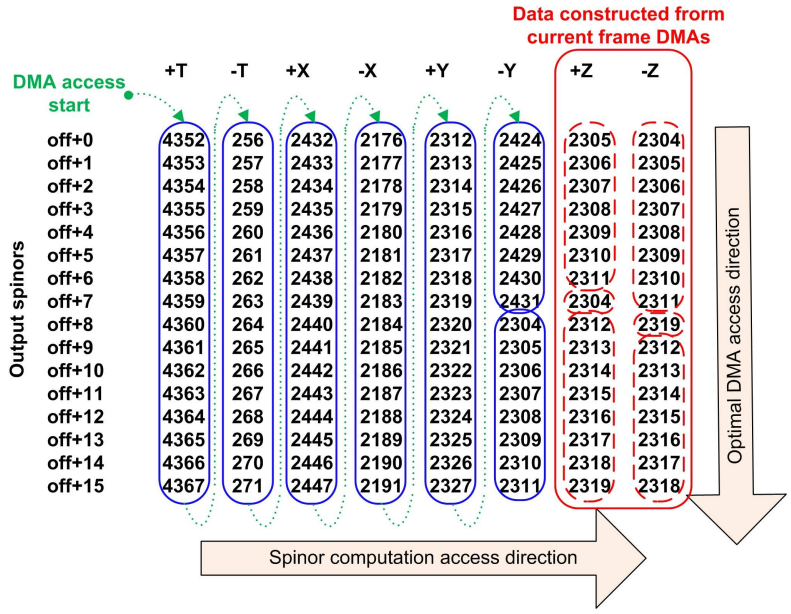


Figure 7: Analysis of access pattern for a frame of spinors.

necessary for computing 64 single precision spinors or 32 double precision spinors. In total, the DMA list carries information about 2048 to 4096 spinors. The eight SPE can hold the optimized DMA information for lattice size $16^3 \times 16$ of single precision (or half of that for double precision).

To handle larger lattice sizes the optimized DMA lists, created in the beginning of the program, can be fetched in pieces. The overhead of this process is minimal since it occurs once every 64 frames of computation and the memory request involves small amount of data. The layout of the data on the local store including the optimized DMA list is shown in Figure 8.

Benefiting from these optimized DMA scheme depends on whether the bandwidth is the performance bottleneck or not. Figure 9 shows the use of double buffering to overlap computation with communication. For single precision computation on Cell BE, the DMA performance is in the critical path of execution time. For double precision, the computations hide the latency for DMA completely. For Cell EDP, simulations show that the wait for DMA will be put back in the critical path of execution for double precision computations. The optimized DMA, shown on the right of Figure 9, reduces the stress on the DMA and introduces additional work to the computing thread. The overall execution time can be lowered if DMA is bounding the performance.

The computation of the optimized DMA by the PPE is outlined by Algorithm 1. Contiguous spinors within a single frame are grouped into single DMA (if their size does not

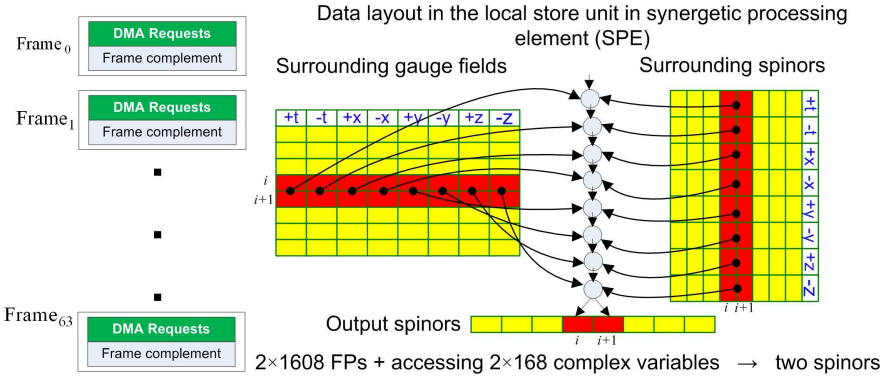


Figure 8: Data layout inside the local store of a Cell SPE. Figure shows 2-way fusion access pattern.

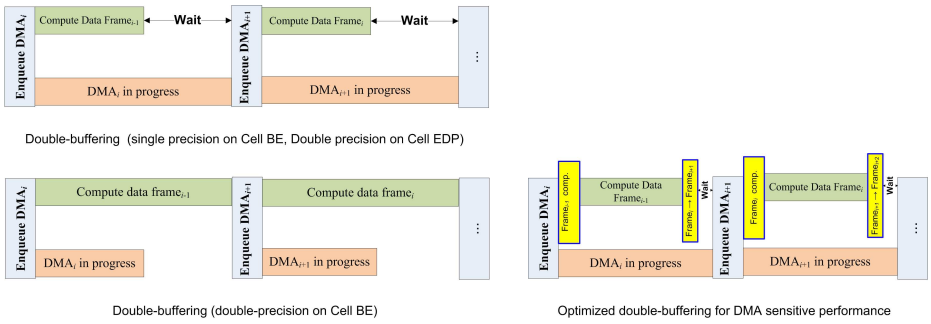


Figure 9: Double buffering for computing spinors in single-precision and double-precision computations. Optimized double-buffering is also shown on the right based on frame analysis.

**Algorithm 1** CREATE_FRAME_DMA(), Compute frame DMA and copy list (PPE side computation).

$offsets \leftarrow$ COMPUTE_FRAME_OFFSETS()
$dmas \leftarrow$ CREATE_CONTIGUOUS_DMA($offsets$)
$(minimal\_dma, copy\_list) \leftarrow$ CREATE_FRAME_COPY_LIST($dmas$)
**return** $(minimal\_dma, copy\_list)$

**Algorithm 2** Frame analysis to create optimized DMA and copy list (PPE side computation).

$prev\_dma \leftarrow$ CREATE_FRAME_DMA()
$spe[c].dmas[0] \leftarrow prev\_dma$
**for** $j \leftarrow 1$ to $dma\_per\_spe - 1$ **do**
  $current\_dma \leftarrow$ CREATE_FRAME_DMA()
5: $temp\_dma \leftarrow current\_dma$
  $current\_dma \leftarrow$ INTER_FRAME_ANALYSIS($prev\_dma, current\_dma$)
  $spe[c] \cdot dmas[j] \leftarrow current\_dma$
  $prev\_dma \leftarrow temp\_dma$
**end for**

exceed 16KB, otherwise they are split). DMAs are then checked for redundancy, when detected the DMA is removed or split and a copy entry is created. The same analysis is done between frames that are to co-exist within the local store, as outlined in Algorithm 2.

The computation engine with the optimized DMA and the inter-frame analysis is shown in Algorithm 3. The computation uses double buffering with the optimized DMA to overlap computation and communication whenever possible.

## 5.2 Performance with DMA

In this section, we present the performance based on 4-way fusion for single precision and 2-way fusion for double precision. We also report results estimated by the simulator for future generation Cell EDP with enhanced double precision performance.

We show the performance with three implementations of the DMA

- DMA with no optimization: The row major layout for spinors is not changed thus causing fragmentation for DMA.
- DMA with contiguity: The spinor data in the local store are aligned in column major format and the DMAs are requested form memory for all data on contiguous fashion when possible.

- Optimized DMA: only non redundant data are requested from the memory. The data are complemented either from the data brought in the current frame or from the data available in the previous frame.

**Algorithm 3** SPE computation of spinors with optimized DMA.

```
    REQUEST_OPTIMIZED_DMAS_STRUCTURE(0)
    for i ← 0 to sublattices − 1 do
      phase ← 0
      WAIT_OPTIMIZED_DMAS_STRUCTURE(i)
 5:   ISSUE_FRAME_INPUTS_DMAS(phase,0)
      phase ← phase ⊕ 1
      for k ← 0 to frames_per_spe − 2 do
        ISSUE_FRAME_INPUTS_DMAS(phase, k + 1)
        phase ← phase ⊕ 1
10:     WAIT_INPUTS_RECIEVED(phase)
        COMPLEMENT_TRANSFER(k)
        WAIT_OUTPUTS_SENT(phase)
        KERNEL_COMPUTE(phase)
        TRANSFER_FROM_PREV(k + 1)
15:     ISSUE_FRAME_WRITE_DMA (phase)
      end for
      phase ← phase ⊕ 1
      WAIT_INPUTS_RECIEVED(phase)
      COMPLEMENT_TRANSFER(k)
20:   WAIT_OUTPUTS_SENT(phase)
      if i + 1 < sublattices then
        REQUEST_OPTIMIZED_DMAS_STRUCTURE(i + 1)
      end if
      KERNEL_COMPUTE(phase)
25:   ISSUE_FRAME_WRITE_DMA (phase)
    end for
    WAIT_OUTPUTS_SENT(phase)
```

Figure 10 shows the execution time breakdown for the three DMA schemes on Cell BE and Cell EDP.

Enqueuing unoptimized DMA requests consumes large percentage of execution time ranging between 50% and 90%. The main reason for that large delay is that only 16 DMA requests are allowed at a time. Exceeding this limit causes the SPE execution to stall waiting for the completion of DMAs and thus reducing the chance of overlapping computation with communication. Another problem with fragmented DMA is that it is not always aligned on 128 bytes boundary of address space. The fragmentation in the single precision computation

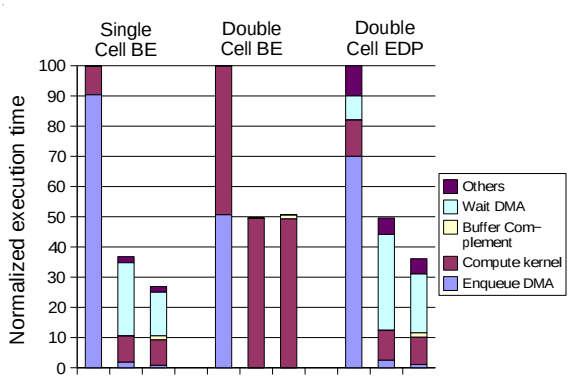


Figure 10: Execution time breakdown with DMA for fragmented DMA, contiguous DMA, and optimized DMA. Each group is normalized to the fragmented version.

is more severe compared with double precision. Aligning data to 128 bytes boundary would reduce the number of spinors that can be computed per frame to half.

Contiguous DMA requests created by observing contiguity on spinors accessed in the same spinor of the space. Enqueuing time becomes negligible and the wait time for DMA show up as a limiting factor for single precision on Cell BE with about 62% of the execution time and is predicted for double precision on future Cell EDP to be 66% of the execution time. The double precision performance on Cell BE is not affected by the latency of the DMA.

The optimized DMA reduces the stall for DMA by 30-40%. Buffer repair (complement) after DMA consumes less than 5% of the execution time for single precision on Cell BE and double precision on Cell EDP. For double precision on Cell BE, the buffer repair takes 2.5% of the execution time.

Table 3 summarizes the performance in GFlops and the bandwidth needed in GB/s. Compared with contiguous DMA, optimized DMA achieves 37% performance improvement for single precision. Future Cell EDP is expected to observe 39% performance improvement for optimizing DMA.

For double precision on Cell BE, the best performance is achieved with contiguous DMA without optimization because copying spinors is added to the critical path of execution. The reduction in performance is about 2%, while almost 26% of the bandwidth is saved which potentially can save power consumption.

The improvement in performance is higher than the saving in bandwidth because other savings are associated with reducing the requested data such as reducing the queuing delay, reducing fragmentation repairs, and reducing the controller occupancies.

| | Single on Cell | | | Double on Cell | | | Double on EDP | | |
|---|---|---|---|---|---|---|---|---|---|
| Effective | frag. | contig. | opt. | frag. | contig. | opt. | frag. | contig. | opt. |
| GFlops | 8.4 | 22.81 | 31.2 | 4.34 | 8.75 | 8.56 | 6 | 11.95 | 16.6 |
| GB/s | 7.86 | 21.33 | 23.58 | 7.78 | 15.68 | 12.4 | 10.74 | 21.6 | 24.05 |

Table 3: Performance of Wilson-Dirac operator routine in terms of GFlops and Bandwidth requirements

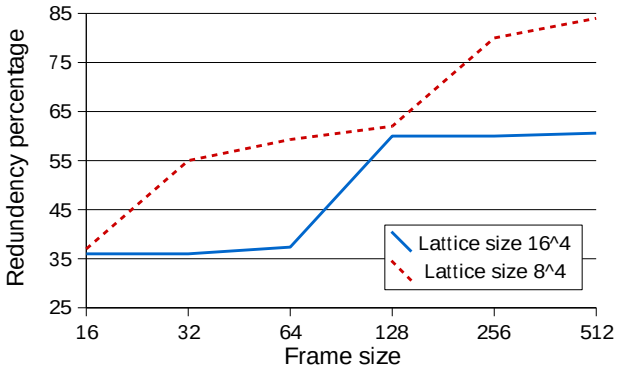


Figure 11: Contiguous redundancy within a frame of spinors.

Another observation is that the saving in bandwidth is different for single precision (24%) compared with double precision (26%). The bandwidth saving is dependent on the fragmentation of data and the amount of redundancy within a frame. The fragmentation for double precision frame is lower compared with single precision frame because of the larger size of data structures for the spinors and the gauge field. On the other hand, the redundancy within a frame is dependent on the frame size where for single precision it is 64 while it is 32 for double precision. The bigger the frame size the better the chance to find redundancy of spinors. If the local store size is increased for the future generation Cell then we expect to have higher saving in bandwidth based on our optimized DMA. Figure 11 shows the increase in the amount of redundancy within a frame with the increase of frame size. Increasing the lattice size also requires increasing frame size to detect the same amount of redundancy.

The simulations we conducted for Cell EDP conservatively assumed that the local store size (and thus the frame size) will be the same as the current generation Cell BE. If the local store size increases, as expected, then the effectiveness of optimizing DMA, by removing redundancy, is expected to increase.

# 6 SPEs Utilization

The SPEs are fully utilized for double precision computation on the current generation Cell BE. For single precision computation on Cell BE and double precision on Cell EDP, the processor stalls for large percentage of the execution cycles.

The main reason is that the cycles that the memory controller will be fully occupied bringing one frame of data, based on 25.6 GB/s, is not less than 11.5 KCycles. Having eight SPEs per Cell BE, each SPE will have 92.2 KCycles to do computation. These cycles are about 3 times the cycles needed for computing a single precision frame on Cell BE. A higher ratio is predicted for double precision on Cell EDP, if the memory bandwidth is not improved. This shows why the SPE will be underutilized in these cases.

Figure 12 shows the performance achieved by partial use of the Cell BE computational resources. The experiments were done on IBM Blade Center® QS20 system varying the used SPE from 1 to 8. For single precision calculation, it is apparent that as few as four SPEs can achieve most of the performance the Cell processor can afford, considering 4-way fusion. The performance slows down by 5% if we increase the SPEs from four to eight because we use more contenting SPEs on the limited bandwidth. For single precision computation, we achieve the same maximum throughput (defined by the bandwidth) for 2-way fusion and 4-way fusion. For single precision 2-way fusion, we achieve the maximum throughput with 6 SPEs compared with 4 SPEs for the 4-way fusion. For double precision 2-way fusion steadily shows better performance compared with no fusion.

Although the flattening throughput is disappointing, it allows an additional degree of freedom for designing a multi-cell system to simulate Lattice QCD. To handle the imbalance of the computational resources with the bandwidth for Lattice QCD on Cell BE, among alternative approaches are

- The memory of some of the SPEs can be used as an extended memory for the other SPEs, for instance, to hold optimized DMA information. The inter-SPE bandwidth is much higher than the bandwidth with the external memory. The Cell processor will then be able to hold information about very large lattice size (for instance $32^3 \times 64$).
- In addition to the computing SPE, other SPE can manage their local storage as additional buffers for frames. Creating extended frames increases the chance of detecting redundancy within the frames and thus reducing the pressure on the system bandwidth.
- Few SPE can be used and others are turned off to save power, or computing threads migrate between SPEs to reduce hotspoting parts of the Cell processor. The runtime fusion can be be viewed as power-saving optimization.

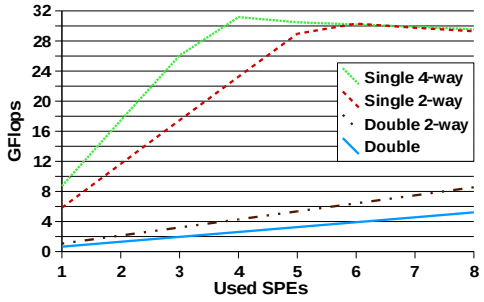


Figure 12: Scalability of performance with the count of used SPEs.

# 7 Conclusions

Porting the computation of the actions of Wilson-Dirac Operators on cell broadband engine requires two special program design processes; the first process is SIMDizing the code to achieve good performance on this architecture; the second process involves optimizing DMA requests.

For Wilson-Dirac operator, no single compile time data layout can be optimal for SIMDizing this code. In this work, we presented the "runtime data fusion" model to overcome the difficulty in static fusion and to reduce the need to shuffle operations. We show that we can achieve 79.6 GFlops for single precision computation and 8.8 GFlops for double precision (50 GFlops for double precision is expected on future Cell EDP).

Considering DMA without analysis makes the observable performance no more than 8.4 GFlops for single precision and 4.3 GFlops for double precision (6 GFlops on Cell EDP).

We show that analysis of frames of data can help in saving 26% percent of the bandwidth in addition to improving contiguity of access. The PPE undertakes the job of analyzing the data access for contiguity and redundancy once in the beginning of the program and then transfer to the SPEs optimized DMA requests and buffer repairs. With optimized DMA, we observed 31.2 GFlops for single precision, 8.75 GFlops for double precision, and we predict 16.6 GFlops of double precision performance on Cell EDP based on the performance simulator provided by IBM.

# Acknowledgment

We would like to thank the "Centre Informatique National de l'Enseignement Supérieur (CINES), France" for allowing us to access their machines to conduct our experiments. We would like also to thank André Seznec for his insightful comments and suggestions to improve this work.

# References

[1] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi, and K. K. Szabo, "Lattice QCD as a Video Game," *arXiv:hep-lat/0611022v2*, 2007.

[2] K. Z. Ibrahim, F. Bodin, and O. Pene, "Fine-grained Parallelization of Lattice QCD Kernel Routine on GPU," *First Workshop on General Purpose Processing on Graphics Processing Units, Northeastern Univ., Boston*, Oct 2007.

[3] NVIDIA Cuda 1.0, "http://developer.nvidia.com/object/cuda.html," 2007.

[4] F. Belletti, G. Bilardi, M. Drochner, N. Eicker, Z. Fodor, D. Hierl, H. Kaldass, T. Lippert, T. Maurer, N. Meyer, A. Nobile, D. Pleiter, A. Schaefer, F. Schifano, H. Simma, S. Solbrig, T. Streuer, R. Tripiccione, and T. Wettig, "QCD on the Cell Broadband Engine," Oct 2007.

[5] S. Motoki and A. Nakamura, "Development of QCD code on a CELL Machine," *Proceeding of Science*, Oct 2007.

[6] Cell SDK 3.0, "http://www.ibm.com/developerworks/power/cell/index.html," Oct. 2007.

[7] C. Urbach, K. Jansen, A. Shindler, and U. Wenger, "HMC Algorithm with Multiple Time Scale Integration and Mass Preconditioning," *Computer Physics Communications*, vol. 174, p. 87, 2006.

[8] C. Urbach, "Lattice QCD with Two Light Wilson Quarks and Maximal Twist," *The XXV International Symposium on Lattice Field Theory*, 2007.